



2 Berechenbarkeitstheorie

2.1 Intuitiver Berechenbarkeitsbegriff und Churchsche These



2.2 Intuitiver Berechenbarkeitsbegriff

$f : \mathbb{N}^k \rightarrow \mathbb{N}$ sei (u.U. partielle) Funktion.

f ist **berechenbar** gdw.

\exists Rechenverfahren(=Algorithmus) das f **berechnet**

Rechenverfahren = Java-Programm (, ..., "geeignete"
Programmiersprache)

Eingabe: $(x_1, \dots, x_k) \in \mathbb{N}^k$

Ausgabe: $f(x_1, \dots, x_k)$

Programm **stoppt** nach endlich vielen Schritten falls $f(x_1, \dots, x_k) \in$
Definitionsbereich von f .

Endlosschleife sonst.



Berechenbarkeit Hauptergebnis

Sicherstes Rezept Nichtinformatiker zu vergraulen:

Eine hitzige Debatte über **DIE** richtige Programmiersprache

- Alle Programmiersprachen und Maschinenmodelle sind
„gleich mächtig“
- In jedem Modell gibt es die gleichen **nichtberechenbaren** Probleme



Beispiel

input n

repeat

until false

berechnete die total undefinierte Funktion Ω

**Beispiel**

$$f_{\pi}(n) = \begin{cases} 1 & \text{falls } n \text{ ist Anfangsabschnitt der Dezimalbruchentwicklung von } \pi. \\ 0 & \text{sonst} \end{cases}$$

f_{π} ist berechenbar:

Benutze Langzahlarithmetik, und iteriere eine geeignete

Näherungsformel "lange genug".

**Exkurs: einige Näherungen für π**

Archimedes: Näherung über gleichseitige Vielecke.

Altindisch: 1682 von Leibniz wiederentdeckt

$$\pi = 4 \sum_{i=0}^{\infty} \frac{-1^i}{2i+1} = 1 - \frac{1}{3} + \frac{1}{5} - \dots$$

("geeignete Abbruchbedingung")

Baile-Borwein-Plouffe 1996:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i-1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

**Beispiel**

$$f(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der Dezimalbruchentwicklung von } \pi \text{ vorkommt.} \\ 0 & \text{sonst} \end{cases}$$

Es ist **unbekannt** ob f berechenbar ist !

Es ist sogar unbekannt ob $\forall n : f(n) = 1$ ("Normalität" von π).

**Beispiel**

$$f(n) = \begin{cases} 1 & \text{falls } n \times '7' \text{ irgendwo in der Dez.-Entwicklung von } \pi \text{ vorkommt.} \\ 0 & \text{sonst} \end{cases}$$

Is f berechenbar ?

**Beispiel**

$$f(n) = \begin{cases} 1 & \text{falls } n \times '7' \text{ irgendwo in der Dez.-Entwicklung von } \pi \text{ vorkommt.} \\ 0 & \text{sonst} \end{cases}$$

Is f berechenbar ? **Ja !**

Fall $\forall n : n \times '7'$ kommt vor: $\forall n : f(n) = 1$

Fall n kommt höchstens $n_0 \times$ irgendwo vor:

$$\longrightarrow f(n) = \begin{cases} 1 & \text{falls } n \leq n_0 \\ 0 & \text{sonst} \end{cases}$$

Also: Berechenbarkeit \neq wir können die Funktion definitiv angeben !

**Beispiel**

LBA: linearbeschränkte Turingmaschine

DLBA: deterministische linearbeschränkte Turingmaschine

$$i(n) = \begin{cases} 1 & \text{falls } \forall LBA M \exists DLBA D : L(M) = L(D) \\ 0 & \text{sonst} \end{cases}$$

Wir wissen nicht wie $i(n)$ aussieht.

Aber, $i(n)$ ist eine konstante Funktion und damit offensichtlich **berechenbar**.

**Nichtberechenbare Funktionen**

Sei $r \in \mathbb{R}$ beliebig.

$$f_r(n) = \begin{cases} 1 & \text{falls } n \text{ ist Anfangsabschnitt der Dezimalbruchentwicklung von } r. \\ 0 & \text{sonst} \end{cases}$$

Annahme: $\forall r : f_r$ ist berechenbar.

$\longrightarrow \exists$ mindestens soviele berechenbare Funktionen wie reelle Zahlen.

Widerspruch:

- Die Menge der berechenbaren Funktionen ist abzählbar
(weil durch endlich langen Text beschrieben).
- \mathbb{R} ist nicht abzählbar.

**Nichtberechenbare Funktionen**

Es gibt sie. Aber können wir eine konkret hinschreiben ?
todo



Church'sche These

Von Turingmaschinen berechenbare Funktionen sind genau die im intuitiven Sinne berechenbaren Funktionen.

Kein Satz aber allgemein akzeptiert.

Begründung

- Alle bekannten Berechnungsmodelle selbst sind schwächer oder äquivalent.
das kann man beweisen
- Keine „intuitiv“ berechenbare Funktion bekannt, die nicht Turing-berechenbar ist.



Turingmaschinen berechnen Funktionen

$T = (Q, \Sigma, \Gamma, \delta, s, F)$ realisiert die partielle Funktion

$$f_T : \Sigma^* \rightarrow \Gamma^* \Leftrightarrow$$

$$f_T(w) := \begin{cases} v & \text{falls } T \text{ hält nach Eingabe von } w \text{ mit Ausgabe } v \\ ((s)w \Rightarrow u(q)v), q \in F \text{ (Schöning: } u \text{ weglassen)} \\ \perp = \text{(undefiniert)} & \text{sonst} \end{cases}$$

g Turingberechenbar $\Leftrightarrow \exists T : f_T = g$

Bemerkung: wenn $g(x) = \perp$ hält T nicht.



Turingmaschinen berechnen numerische Funktionen

$f : \mathbb{N}^k \rightarrow \mathbb{N}$ Turingberechenbar \Leftrightarrow

$\exists T = (Q, \Sigma, \Gamma, \delta, s, F) : \forall n_1, \dots, n_k, m \in \mathbb{N} :$

$$f(n_1, \dots, n_k) = m \Leftrightarrow$$

$$(s)\text{bin}(n_1)\#\dots\#\text{bin}(n_k) \vdash^* u(q)\text{bin}(m), q \in F$$



Akzeption \rightsquigarrow Funktion

Funktionenberechnung ist der allgemeinere Begriff.

Statt Akzeptor für $L \subseteq \Sigma^*$ betrachte TM, die die „halbe“ charakteristische Funktion

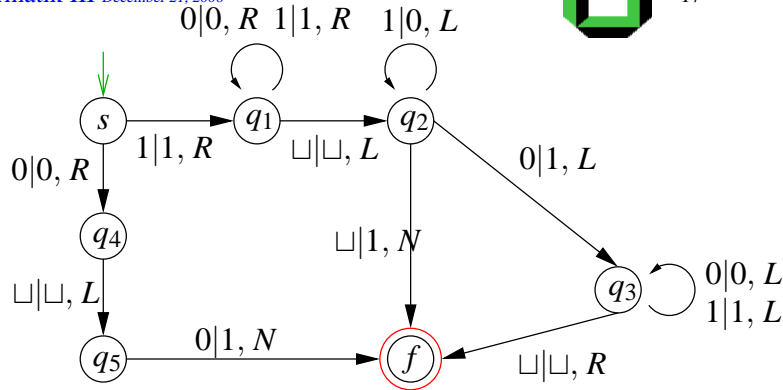
$$\chi'_L(w) = \begin{cases} 1 & \text{falls } w \in L \\ \perp & \text{sonst} \end{cases}$$

berechnet.

Aber, wie gesagt, alles „interessante“ kann man bereits mit Akzeptoren machen.



Beispiel



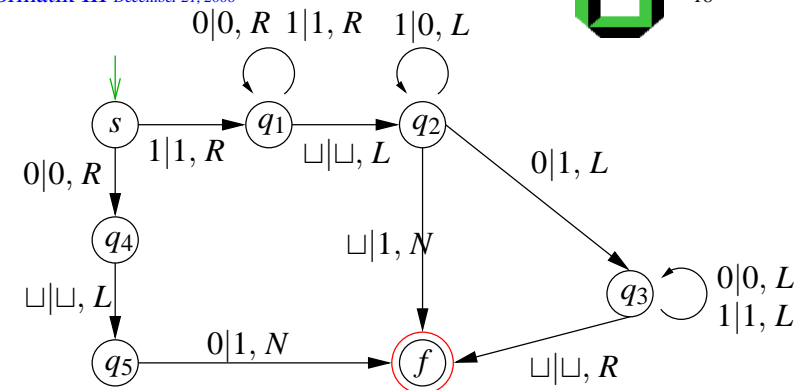
$$f(w) = \begin{cases} w + 1 & \text{falls } w \in 0 \cup 1(0 \cup 1)^*, \\ & w \text{ interpretiert als Binärzahl} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Bemerkung: Nichteingezeichnete Übergänge gelten hier als

Endlosschleife.



Beispiel



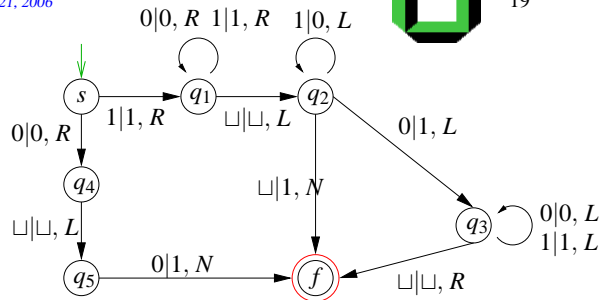
$$(s)11 \vdash 1(q_1)1 \vdash 11(q_1) \vdash 1(q_2)1 \vdash (q_2)10 \vdash (q_2)\square 00 \vdash (f)100$$



Funktionsweise

Fallunterscheidung nach Aufbau der Eingabe.

Sei $w \in \{0, 1\}^*$ beliebig,
 $a \in \{0, 1\}, n \geq 1$.



$$0: (s)0 \vdash 0(q_4) \vdash (q_5)0 \vdash (f)1$$

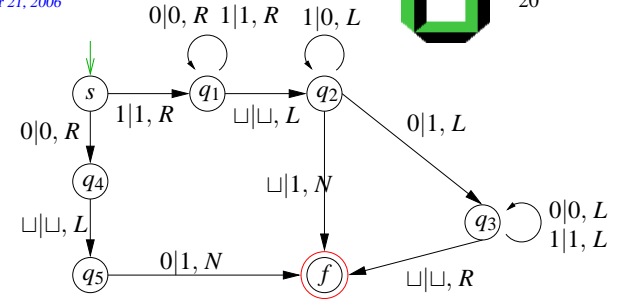
$$0aw: (s)0aw \vdash 0(q_4)aw \text{ nichtdefiniert}$$



Funktionsweise

Fallunterscheidung nach Aufbau der Eingabe.

Sei $w \in \{0, 1\}^*$ beliebig,
 $a \in \{0, 1\}, n \geq 1$.



$$1^n: (s)1^n \vdash 1(q_1)1^{n-1} \vdash 1^n(q_1) \vdash 1^{n-1}(q_2)1 \vdash (q_2)\square 0^n \vdash (f)10^n$$

$$1w0: (s)1w0 \vdash 1(q_1)w0 \vdash 1w0(q_1) \vdash 1w(q_2)0 \vdash 1w(q_3)1 \vdash (q_3)\square 1w1 \vdash (f)1w1$$

$1w01^n:$

$$(s)1w01^n \vdash 1(q_1)w01^n \vdash 1w01^n(q_1) \vdash 1w01^{n-1}(q_2)1 \vdash 1w(q_2)00^n \vdash 1w(q_3)10^n \vdash (q_3)\square 1w10^n \vdash (f)1w10^n$$



Beispiel: Die überall undefinierte Funktion

$$T = (\{s\}, \Sigma, \Gamma, \delta, s, \{\})$$

$$\forall a \in \Gamma : \delta(s, a) = (s, a, R)$$



Lokale Variablen

Lokale Variable $x \in A$ speichern, ($|A| < \infty$!):

$$Q \rightsquigarrow Q \times A$$



Programmiertechniken für Turingmaschinen

- Lokale Variablen
- Hintereinanderschalten
- Spuren
- While-Schleifen



Hintereinanderschalten

Gegeben: $T = (Q, \Sigma, \Gamma, \delta, s, F)$

OBdA: $(s)w \vdash^*(r)f_T(w)$ für ein $r \in F$ falls $f_T(w) \neq \perp$,

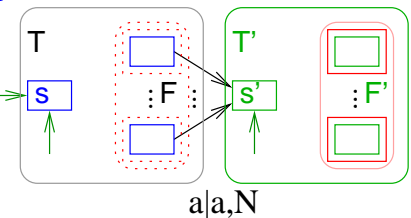
$T' = (Q', \Sigma, \Gamma', \delta', s', F')$.

Ausgabe: Turingmaschine $T^\circ = (Q^\circ, \Sigma, \Gamma^\circ, \delta^\circ, s, F')$ für $f_{T'}(f_T(x))$:

$$Q^\circ = Q \cup Q'$$

$$\Gamma^\circ = \Gamma \cup \Gamma'$$

$$\delta^\circ(q, a) = \begin{cases} \delta(q, a) & \text{falls } q \in Q \setminus F \\ (s', a, N) & \text{falls } q \in F \\ \delta'(q, a) & \text{falls } q \in Q' \end{cases}$$





Spuren

$$\Gamma = \Gamma_1 \times \dots \times \Gamma_k.$$

Beispiele: Arithmetische Operationen auf 2 Binärzahlen,

Markierungen...

Kleine Komplikation: Eingabealphabet ändert sich.

Auswege:

- $\Gamma = \Sigma \dot{\cup} \Gamma_1 \times \dots \times \Gamma_k$
- Ersetze $a \in \Sigma$ durch $(a, 0, \dots, 0)$ in der Eingabe.



While-Schleifen: While $i \neq 0$ Do $\text{tape} := f_T(\text{tape})$

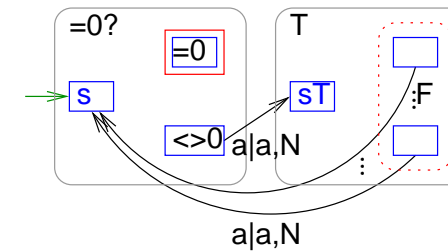
Spur i definiere einen Zähler (unär oder binär)

Unterprogramm: teste ob Spur $i = 0$.

Wenn ja: halt

Lass T laufen

Zurück zum Startzustand. (Übergang $\delta(f, a) = (s, a, N)$)

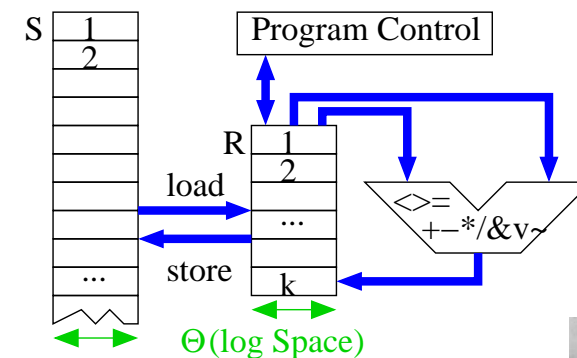


2.3 LOOP-, WHILE-, GOTO (=Registermaschinen) und RAM- Berechenbarkeit

- Mehrere einfache Berechnungsmodelle
- Alle bis auf eines Turing-mächtig



RAM: Random Access Machine

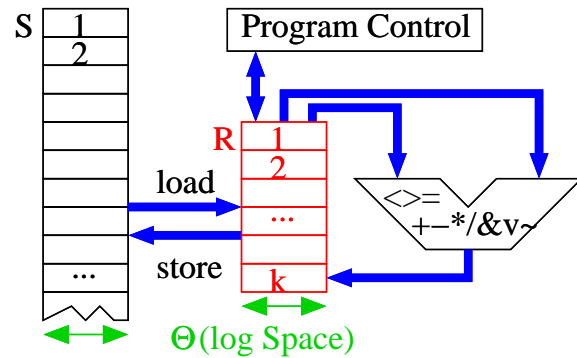


Moderne (RISC) Adaption des
von Neumann-Modells [von Neumann 1945]





Register



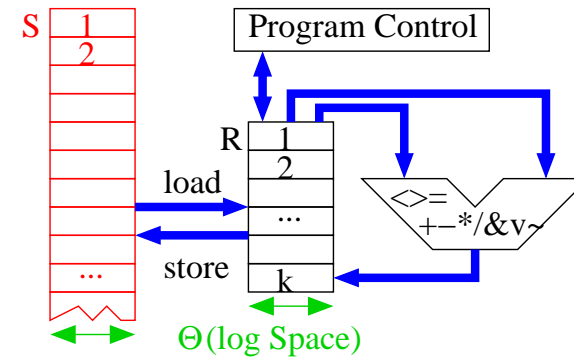
k (irgendeine Konstante) Speicher

R_1, \dots, R_k für

(kleine) ganze Zahlen



Hauptspeicher



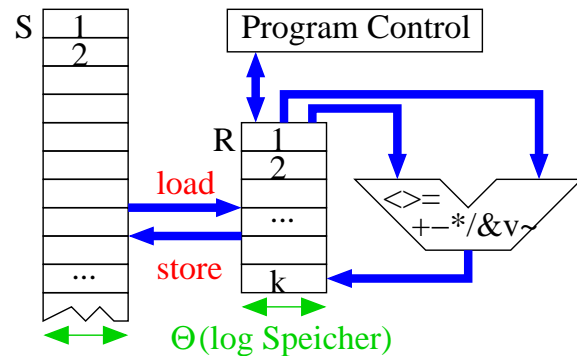
Unbegrenzter Vorrat an Speicherzellen

$S[1], S[2] \dots$ für

(kleine) ganze Zahlen



Speicherzugriff

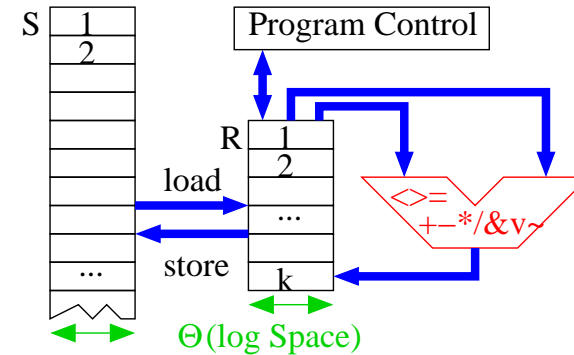


$R_i := S[R_j]$ lädt Inhalt von Speicherzelle $S[R_j]$ in Register R_i .

$S[R_j] := R_i$ speichert Register R_i in Speicherzelle $S[R_j]$.



Rechnen



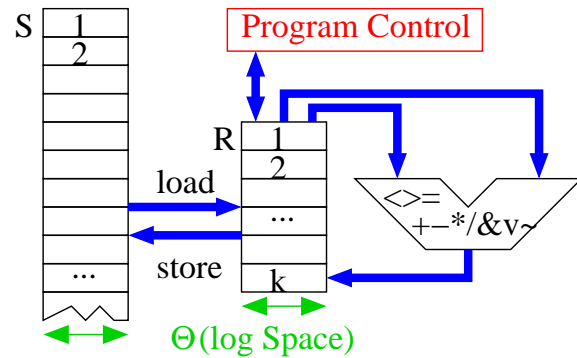
$R_i := R_j \odot R_\ell$ Registerarithmetik.

' \odot ' ist Platzhalter für eine Vielzahl von Operationen

Arithmetik, Vergleich, Logik



Bedingte Sprünge



JZ j, R_i Setze Programmausführung an Stelle j fort falls $R_i = 0$



„Kleine“ ganze Zahlen?

Alternativen:

Konstant viele Bits (64?): theoretisch unbefriedigend weil nur endlich viel Speicher adressierbar \rightsquigarrow endlicher Automat

Beliebige Genauigkeit: viel zu optimistisch für vernünftige

Komplexitätstheorie. Beispiel: n maliges quadrieren führt zu einer Zahl mit $\approx 2^n$ bits.

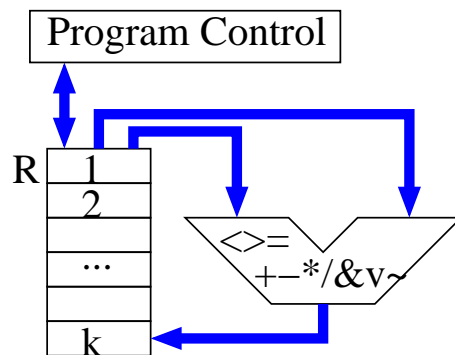
OK für Berechenbarkeit

Genug um alle benutzten Speicherstellen zu adressieren: Bester Kompromiss.



Registermaschine

\approx RAM – Speicher + beliebige Genauigkeit



ggf. eingeschränkt auf Inkrementieren, Dekrementieren

Beliebt in der Berechenbarkeit.

Führt zu merkwürdigen Algorithmen.



Registermaschinen-Berechenbarkeit

Konfiguration: (q, R_1, \dots, R_k)

q ist ein Zustand (z.B. Programmzähler)

„ \vdash^* “ definiert wie gehabt.

$f : \mathbb{N}^{k'} \rightarrow \mathbb{N}, k' \leq k$ Registermaschinen-berechenbar \Leftrightarrow

$\exists \text{RM } M : \forall n_1, \dots, n_{k'}, m \in \mathbb{N} :$

$f(n_1, \dots, n_{k'}) = m \Leftrightarrow$

$(1, n_1, \dots, n_{k'}, 0^{k-k'}) \vdash^*(q, f(n_1, \dots, n_k), \dots)$

mit $\text{PROGRAM}[q] = \text{HALT}$



RAM-Berechenbarkeit

Konfiguration: (q, R_1, \dots, R_k, S)

Sei M eine RAM:

Eingabe: $w \in \Sigma^n$ in $S[1], \dots, S[n]$

Ausgabe: $f_M(w)$ in $S[1], \dots, S[|f_M(w)|]$

wenn HALT-Befehl ausgeführt wird.

Natürliche Zahlen passen i.allg. nicht in einzelne

Register/Speicherzellen und müssen codiert werden ! Analog TM



Höhere Programmiersprachen

Java, C/C++, Pascal, ...

ML, Lisp, ...

Prolog, Oz, ...

...

sind das uns am meisten geläufige Programmiermodell.

Compiler übersetzen das routinemäßig in RAM Code.



LOOP-Programme

Minimalistische Programmiersprache für Berechenbarkeitstheorie:

```

 $\mathbb{N}$  main( $\mathbb{N}x_1, \dots, \mathbb{N}x_k$ ){
   $\mathbb{N} x_0 = 0; \mathbb{N} x_{k+1} = 0; \mathbb{N} x_{k+2} = 0; \dots$ 
  body;
  return  $x_0$ ;
}
    
```

body darf benutzen

Zuweisung: $x_i := x_j + c, c \in \{-1, 0, 1\}$ $0 - 1 := 0$

Schöning: $c \in \mathbb{Z}$

‘;’: Sequenz von Anweisungen

loop x_i : Schleife. Wiederhole x_i mal. Relevant ist der Inhalt von x_i VOR der ersten Schleifenausführung.



LOOP-Programme

Beobachtung: Loop-Programme terminieren immer.

Definition: f ist **Loop-berechenbar** gdw.

\exists Loop-Programm P , das f berechnet.

Frage: welche Funktionen sind Loop-berechenbar?

Gibt es totale berechenbare Funktionen, die **nicht** Loop-berechenbar sind?



Leicht nachzubauende Anweisungen

$x := x + c$ $// c \in \mathbb{Z}$
 $x := y + z$
 $x := y \dot{-} z$ $// y \dot{-} z = y - z$ falls $y \geq z$, 0 sonst
 $x := y \cdot z$
 $x := y \text{ div } z$
 $x := y \text{ mod } z$
beliebige Arithmetische Ausdrücke
if $x \neq 0$ **then** ...



Beispiel Addition $x_0 := x_1 + x_2$

$x_0 := x_1$
loop x_2
 $x_0 ++$



Beispiel Multiplikation $x_0 := x_1 \cdot x_2$

loop x_1
 $x_0 := x_0 + x_2$



Beispiel if $x = 0$ then A

$y := 1$
loop x
 $y --$
loop y
 A



While-Programm

Minimalistische Programmiersprache für Berechenbarkeitstheorie:

```

 $\mathbb{N}$  main( $\mathbb{N}x_1, \dots, \mathbb{N}x_k$ ){
   $\mathbb{N} x_0 = 0; \mathbb{N} x_{k+1} = 0; \mathbb{N} x_{k+2} = 0; \dots$ 
  body;
  return  $x_0$ ;
}
    
```

body darf benutzen

Zuweisung: $x_i := x_j + c, c \in \{-1, 0, 1\}$ $0 - 1 := 0$

‘;’: Sequenz von Anweisungen

while($x_i \neq 0$): Schleife



WHILE simuliert LOOP

loop x do

P

~>

$y := x$

// y kommt in P nicht vor

while $y \neq 0$ do

$y := y - 1$

P



Kann LOOP While simulieren ?

Nein !

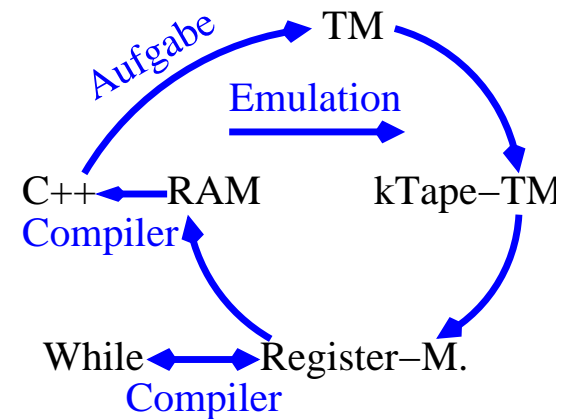
Warum nicht ?

Wenigstens alle totalen Turingberechenbaren Funktionen ?

~> später

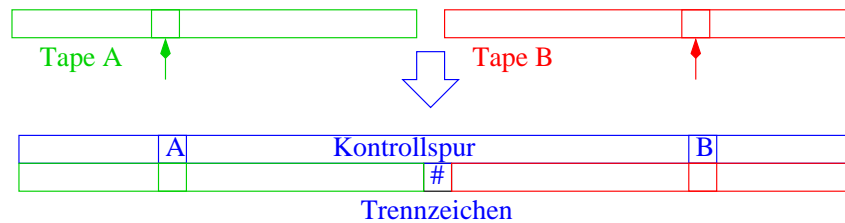


Äquivalenz von Maschinenmodellen





Turingmaschine emuliert k -Band-TM



- Nichleere Bandteile aneinanderhängen (Trennzeichen benutzen)
- Kopfpositionen markieren
- Zustand speichert $k - 1$ Bandsymbole

Satz: Zeit T mit k -Band-TM \rightarrow Zeit $\mathcal{O}(T^2)$ auf Einband-TM.



k -Band-TM emuliert Registermaschine

- Ein Band pro Register (Binärformat oder Unärformat)
- Eigene Zustände für jede Programmzeile
- Unterprogramme für Arithmetik
- Zuweisung \rightarrow Band kopieren



Registermaschine emuliert RAM

Idee: zusätzliches Register R_S repräsentiert Speicher:

$$R_S = \sum_i S[i] \cdot 2^{bi}$$

mit b =Anzahl Bits der RAM

$S[i]$ in R_j laden:

$$R_j := \frac{R_S}{2^{bi}} \bmod 2^b .$$

$S[i] := 0$:

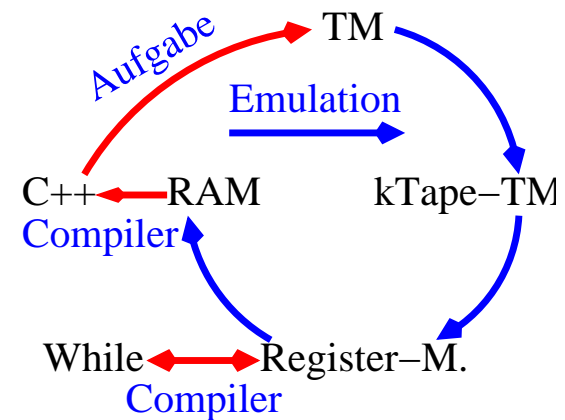
$$R_S := R_S - \left(\frac{R_S}{2^{bi}} \bmod 2^b\right) 2^{bi}$$

R_j in $S[i]$ speichern:

$$S[i] := 0; R_S := R_S + R_j \cdot 2^{bi}$$



Äquivalenz von Maschinenmodellen





Markov-Algorithmen

deterministisches regelbasiertes String-Rewriting.

Geg: Eingabe $w \in \Sigma^*$

Menge von Regeln $\Delta \in (\Gamma^* \times \Gamma^*)^*$

while $\exists (\ell, r) \in \Delta, u, v \in \Gamma^* : w = u\ell v$ **do**

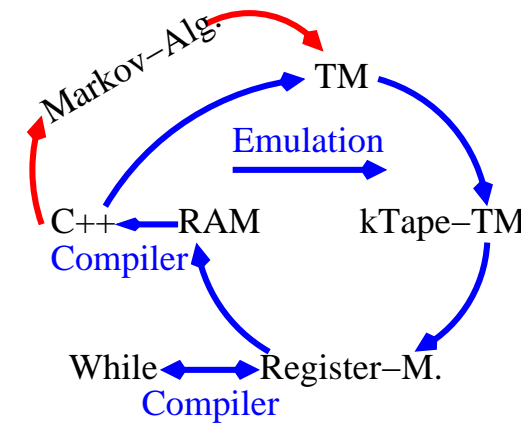
find the first rule $(\ell, r) \in R$

and the shortest $u \in \Gamma^*$ such that $w = u\ell v$ for some $v \in \Gamma^*$

$w := urv$



Markov-Algorithmen: Turingmächtigkeit



Markov-Algorithmen: Turingmächtigkeit

Gegeben: TM $M = (Q, \Sigma, \Gamma, \delta, s, F)$ mit Eingabe w .

OBdA: max 1 \sqcup links und rechts der Eingabe wird angeschaut.

Betrachte Markovalgorithmus für Alphabet $Q \cup \Gamma \cup \{(\cdot, \cdot)\}$.

$\Delta = \dots$ Sonderregeln für den Rand

- $\langle ((q)a, (q')a') : \delta(q, a) = (q', a', N) \rangle$
- $\langle (c(q)a, (q')cd') : \delta(q, a) = (q', a', L) \rangle$
- $\langle ((q)ac, d'(q')c) : \delta(q, a) = (q', a', R) \rangle$

Eingabe des Markovalgorithmus: $\sqcup(s)w\sqcup$

Die Folge der produzierten Strings ist gerade die Folge der **Konfigurationen** der Turingmaschine!



Semi-Thue-Systeme

Sowas wie nichtdeterministische Markov-Algorithmen.

Ebenfalls turingmächtig.

Unsere TM-Simulation hat immer genau eine anwendbare Regel.

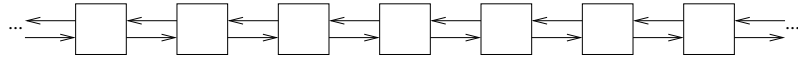


Zellularautomaten

Betrachte den endlichen Automaten $(\{0, 1\}, \{0, 1\}^2, \delta, \emptyset)$ mit

Q	0	1	0	1	0	1	0	1
$L \times R$	(0,0)	(0,0)	(0,1)	(0,1)	(1,0)	(1,0)	(1,1)	(1,1)
δ	0	1	1	1	0	1	1	0

Verbinde unendlich viele solcher Automaten zu einer Kette



[M. Cook 2002]: Diese Maschine ist Turing-mächtig.

siehe auch Wikipedia „rule 110 cellular automaton“



Quantencomputer

- Ein **Qubit** speichert Superpositionen von 0 und 1.
- Berechnungen mit n Qubits berechnen eine Superposition von 2^n klassischen Berechnungen
- Messungen** erhalten bestimmte Informationen über all diese Berechnungen bestimmen
- Quantencomputer können in polynomieller Zeit **faktorisieren** und **diskrete Logarithmen** bestimmen
- Dies würde viele kryptografische Algorithmen kompromittieren



Quantencomputer: Berechenbarkeit und Komplexitätstheorie

- Vermutung der Komplexitätstheorie:
 - Faktorisieren, DLog sind **nicht in P** (selbst mit Randomisierung)
 - Faktorisieren, DLog sind nicht NP hart.
- Turingmaschinen können Quantencomputer simulieren

Fazit: Quantencomputer wären schneller aber nicht mächtiger als klassische Computer



2.4 Primitiv rekursive und μ -rekursive Funktionen

hier nicht



2.5 Die Ackermannfunktion

[Ackermann 1928, Hermes]

Function $a(x, y)$

```

if  $x = 0$  then return  $y + 1$ 
if  $y = 0$  then return  $a(x - 1, 1)$ 
return  $a(x - 1, a(x, y - 1))$ 
    
```



Totalität der Ackermannfunktion

Beweis: Induktion über die **lexikographische Reihenfolge** von (x, y) :

Induktionsanfang: $a(0, y) = y + 1$

Induktionsschritt für $y = 0$:

$a(x, 0) = a(x - 1, 1)$,

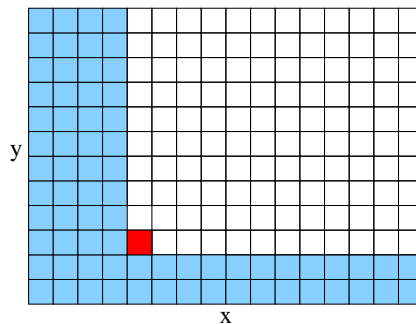
terminiert, da $(x - 1, 1) < (x, 0)$

Induktionsschritt für $x, y > 0$:

$a(x - 1, a(x, y - 1))$ terminiert da

$(x, y - 1) < (x, y)$ und

$(x - 1, a(x, y - 1)) < (x, y)$



Satz: a ist eine totale, TM-berechenbare Funktion

Beweisskizze:

Rekursion \rightsquigarrow Stapel bei RAM \rightsquigarrow TM



Wie große Zahlen berechnet ein Loop Programm?

Definition:

Seien für ein Loop Programm P

$\mathbf{x} = (x_0, x_1, \dots)$ der Variablenvektor bei Programmstart. Hier beliebig!

$\mathbf{x}' = (x'_0, x'_1, \dots)$ der Variablenvektor bei Programmende.

$$f_P(\mathbf{x}) := \sum_{i \geq 0} x'_i$$

$$f_P(n) := \max \left\{ f_P(\mathbf{x}) : \sum_{i \geq 0} x_i \leq n \right\}$$



Die Ackermann Funktion ist **nicht** Loop-berechenbar

Beweisansatz: Annahme a ist Loop-berechenbar.

→ $a(n, n) = g(n)$ ist berechenbar durch ein Loop-Programm G .

→ $a(n, n) = g(n) \leq f_G(n)$

Wir aber zeigen:

Lemma E: \forall Loop-Programm $P : \exists k : \forall n \in \mathbb{N} : f_P(n) < a(k, n)$.

Widerspruch.



Beispiele

$$a(0, y) = y + 1$$

$$\begin{aligned} a(1, y) &= a(0, a(1, y - 1)) = a(1, y - 1) + 1 = \\ &= a(0, a(1, y - 2)) + 1 = a(1, y - 2) + 2 = \dots = \\ &= a(1, 0) + y = y + a(0, 1) = y + 2 \end{aligned}$$

$$\begin{aligned} a(2, y) &= a(1, a(2, y - 1)) = 2 + a(2, y - 1) = \dots \\ &= 2y + a(2, 0) = 2y + a(1, 1) = 2y + 3 \end{aligned}$$



Beispiele

$$a(2, y) = 2y + 3$$

$$\begin{aligned} a(3, y) &= a(2, a(3, y - 1)) = 2a(3, y - 1) + 3 \\ &= 2a(2, a(3, y - 2)) + 3 = 4a(3, y - 2) + 3(1 + 2) \\ &= 4a(2, a(3, y - 3)) + 3(1 + 2) = 8a(3, y - 3) + 3(1 + 2 + 4) \\ &= \dots = 2^y \underbrace{a(3, 0)}_{=5} + 3 \underbrace{(1 + 2 + \dots + 2^{y-1})}_{=2^y - 1} \\ &= 2^{y+3} - 3 \end{aligned}$$



Beispiele

$$a(3, y) = 2^{y+3} - 3$$

$$\begin{aligned} a(4, y) &= a(3, a(4, y - 1)) = 2^{a(4, y - 1) + 3} - 3 \\ &= 2^{a(3, a(4, y - 2)) + 3} - 3 = 2^{2^{a(4, y - 2) + 3} - 3 + 3} - 3 \\ &= 2^{2^{a(3, y - 3) + 3} - 3} - 3 = 2^{2^{2^{a(4, y - 3) + 3} - 3 + 3} - 3} - 3 \\ &= \dots = 2^{\dots} - 3 = 2^{\dots} - 3 \end{aligned}$$

$\underbrace{a(3, 1) = 2^{1+3} - 3}_{a(4, 0)}$

$\underbrace{\dots}_{2} \dots - 3 = 2^{\dots} - 3$

y-Stapel von 2ern

$$a(4, 2) = 2^{2^{16}} - 3 = 2^{65536} - 3$$



Monotonie der Ackermann Funktion

Lemma A: $y < a(x, y)$

Lemma B: $a(x, y) < a(x, y + 1)$

Lemma C: $a(x, y + 1) \leq a(x + 1, y)$

Lemma D: $a(x, y) < a(x + 1, y)$

Lemma BD: $a(x, y) \leq a(x', y')$ falls $x \leq x'$ und $y \leq y'$

Beweis: Übung. Induktion,...



Lemma E: $\forall \text{Loop-Programm } P : \exists k : \forall n \in \mathbb{N} : f_P(n) < a(k, n).$

Beweis: Induktion über Aufbau des Loop-Programms.

Induktionsanfang:

$f_{\text{LeeresProgramm}}(n) = n < n + 1 = a(0, n).$

$f_{x:=y+c}(n) \leq 2n + 1 < 2n + 3 = a(2, n)$



Induktionsschritt für $P = P_1; P_2$:

Nach IV $\exists k_1, k_2 : f_{P_1}(n) < a(k_1, n) \wedge f_{P_2}(n) < a(k_2, n).$

Sei nun $k_3 = \max\{k_1 - 1, k_2\}$. Es gilt:

$$\begin{aligned} f_P(n) &\leq f_{P_2}(f_{P_1}(n)) && \text{Def. } f_P \\ &< a(k_2, a(k_1, n)) && \text{IV, Monotonie} \\ &\leq a(k_3, a(k_3 + 1, n)) && \text{Monotonie} \\ &= a(k_3 + 1, n + 1) && \text{Def. } a \\ &\leq a(k_3 + 2, n) && \text{Lemma B} \end{aligned}$$



Induktionsschritt für $P = \text{loop } x_i \text{ do } Q$:

OBdA x_i kommt in Q nicht vor (ggf. in neue Var. kopieren).

Nach IV $\exists k : f_Q(n) < a(k, n).$

Sei \mathbf{x} eine Eingabe bei der $f_P(\mathbf{x})$ maximiert wird für $\sum_j x_j \leq n.$

Sei $m \leq n$ der Wert von x_i in \mathbf{x}

$$\begin{aligned} f_P(n) &= f_P(\mathbf{x}) \\ &\leq \underbrace{f_Q(f_Q(\dots f_Q(n-m)\dots))}_{m \text{ mal}} + m && \text{Def. } m \\ &\leq a(k, \underbrace{f_Q(f_Q(\dots f_Q(n-m)\dots))}_{m-1 \text{ mal}}) + m - 1 && \text{IV} \dots \\ &\leq a(k, \underbrace{a(k, \dots a(k, n-m)\dots)}_{m \text{ mal}}) + m - m && \text{IV} \end{aligned}$$



Induktionsschritt für $P = \text{loop } x_i \text{ do } Q$:

$$\begin{aligned}
 f_P(n) &\leq \underbrace{a(k, a(k, \dots a(k, n-m) \dots))}_{m \text{ mal}} \\
 &< \underbrace{a(k, a(k, \dots a(k, a(k+1, n-m) \dots))}_{m-1 \text{ mal}} && \text{Monotonie} \\
 &= \underbrace{a(k, a(k, \dots a(k, a(k+1, n-m+1) \dots))}_{m-2 \text{ mal}} && \text{Def. } a \\
 &= \dots = a(k+1, n-1) && \text{Def. } a \\
 &\leq a(k+1, n) && \text{Monotonie}
 \end{aligned}$$

qed.



Mehr schnell wachsende Funktionen

$$k \mapsto \max \{ f_P(0) : P \text{ ist term. While-Programm mit } k \text{ Instr.} \}$$

Fleißige Biber

$\Sigma(n)$: $\max_{\delta} \# \text{Einsen}$, die auf dem Band stehen nachdem eine DTM $(\{1, \dots, n, Z\}, \emptyset, \{0, 1\}, \delta, 1, \{Z\})$ hält (leere Eingabe).

$S(n)$: $\max_{\delta} \# \text{Zustandsübergänge}$, die eine haltende DTM $(\{1, \dots, n, Z\}, \emptyset, \{0, 1\}, \delta, 1, \{Z\})$ gemacht hat (leere Eingabe).



Wissen über Fleißige Biber

$\Sigma(n), S(n)$ sind totale nichtberechenbare Funktionen.

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	$\geq 4\,098$	$\geq 47\,176\,870$
6	$> 1.29 \cdot 10^{865}$	$> 3 \cdot 10^{1730}$

[<http://www.drb.insel.de/~heiner/BB/>],

[<http://www.logique.jussieu.fr/~michel/ha.html>]



Rekordhalter

n:	6	5	4	3	2	
q/in	0 1	0 1	0 1	0 1	0 1	
A:	B1R F0L;	B1R C1L;	B1R B1L;	B1R Z1L;	B1R B1L	
B:	C0R D0R;	C1R B1R;	A1L C0L;	B1L C0R;	A1L Z1R	
C:	D1L E1R;	D1R E0L;	Z1R D1L;	C1L A1L;		
D:	E0L D0L;	A1L D1L;	D1R A0R;			
E:	A0R C1R;	Z1R A0L;				n=1:
F:	A1L Z1R;					H1N ---



Langsam wachsende Funktionen

Inverse Ackermannfunktion

$$\alpha(m, n) := \min \{i \geq 0 : a(i, \lfloor m/n \rfloor) > \log_2 n\}$$

Für jeden realistischen Fall gilt $\alpha(m, n) \leq 5$.

Aber $\alpha(m, n) \notin O(1)$.

Eine wichtige Datenstruktur hat Gesamtkomplexität $m\Theta(\alpha(m, n))$ für m Operationen auf n Objekten:



Union-Find Datenstruktur

Class UnionFind($n : \mathbb{N}$) // Maintain a partition of $1..n$

Function find($i : 1..n$) : $1..n$

assert $\forall i, j \in \{1, \dots, n\} :$

find(i) = find(j) \Leftrightarrow i, j are in the same part

Procedure union($i, j : 1..n$)

$A :=$ the part with $i \in A$

$B :=$ the part with $j \in B$

join A and B to a single part

Anwendung: z.B. **Kruskal's** Algorithmus für **minimale Spannbäume**



Class UnionFind($n : \mathbb{N}$)

parent = $[1, 2, \dots, n]$: **Array** $[1..n]$ of $1..n$

gen = $[0, \dots, 0]$: **Array** $[1..n]$ of $0.. \log n$ // generation of leaders

Function find($i : 1..n$) : $1..n$

if parent[i] = i **then return** i

else $i' :=$ find(parent[i])

parent[i] := i' // **path compression**

return i'

Procedure link($i, j : 1..n$)

assert i and j are leaders of different subsets

if gen[i] < gen[j] **then** parent[i] := j // balance

else parent[j] := i ; **if** gen[i] = gen[j] **then** gen[i]++

Procedure union($i, j : 1..n$)

if find(i) \neq find(j) **then** link(find(i), find(j))

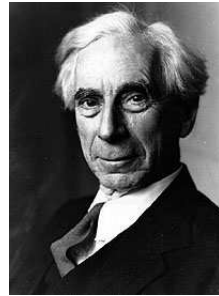


2.6 Halteproblem, Unentscheidbarkeit, Reduzierbarkeit

- Gödelnummerierung: TMs können sich selbst als Eingabe verarbeiten
- Wichtiges Beispiel: Universelle TM
- Diagonalisierungsargument: definiere unentscheidbare Sprache
- Reduktionen: Zeige, dass auch **nützliche** Probleme unentscheidbar sind



Paradoxien und Selbstbezüglichkeit



Der Barbier von Hintertupfingen
rasiert genau die Männer im Dorf,
die sich nicht selbst rasieren.

Wer rasiert den Barbier?



Paradoxien und Selbstbezüglichkeit

Daniel Düsentrieb behauptet, eine allwissende Maschine erfunden zu haben.

Ja Nein

Man stellt eine Ja/Nein-Frage und die Antwort leuchtet auf.

Dagobert Duck kauft die Maschine.

Will aber nur bei korrekter Funktion zahlen.

Er stellt der Maschine die Frage:

Wirst Du mit **Nein** antworten?

Was passiert?



Entscheidbarkeit

$A \subseteq \Sigma^*$ entscheidbar gdw.

die **charakteristische Funktion** χ_A berechenbar ist.

$$\chi_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ 0 & \text{falls } w \notin A \end{cases}$$



Semi-Entscheidbarkeit

$A \subseteq \Sigma^*$ **semi**-entscheidbar gdw.

die „**halbe**“ charakteristische Funktion χ_A berechenbar ist.

$$\chi_A(w) = \begin{cases} 1 & \text{falls } w \in A \\ \perp & \text{falls } w \notin A \end{cases}$$



Satz: $A \subseteq \Sigma^*$ entscheidbar \Leftrightarrow

A und \bar{A} sind beide semi-entscheidbar

Beweisskizze: Sei TM

M_A Akzeptor für A und

$M_{\bar{A}}$ Akzeptor für \bar{A}

for $s := 1$ **to** ∞ **do**

if M_A stoppt nach s Schritten **then** Accept

if $M_{\bar{A}}$ stoppt nach s Schritten **then** Reject



Rekursive Aufzählbarkeit

$A \subseteq \Sigma^*$ rekursiv aufzählbar gdw.

\exists totale berechenbare Funktion $f : \mathbb{N} \rightarrow \Sigma^*$:

$$A = \{f(1), f(2), f(3), \dots\}$$

Satz: A ist rekursiv aufzählbar $\Leftrightarrow A$ ist semi-entscheidbar



Rekursiv aufzählbar \longrightarrow semi-entscheidbar

Sei A rekursiv aufzählbar mittels f .

Function $\chi'_A(x)$

for $s := 1$ **to** ∞ **do**

if $f(n) = x$ **then return** 1



Semi-entscheidbar \longrightarrow rekursiv aufzählbar

Fall $A = \emptyset$: trivial.

Sonst geben wir eine Funktion $f : \mathbb{N} \rightarrow \Sigma^*$ mit Wertebereich A an.

Function $f(n)$

$a :=$ some fixed element of A

 interpret n as bit string w

 split w into w_1, w_2 with $|w_1| = \lceil |w|/2 \rceil, |w_2| = \lfloor |w|/2 \rfloor$

 interpret w_1 as an integer k

 interpret w_2 as a word $u \in \Sigma^*$

if an acceptor M for A accepts u in $\leq k$ steps **then return** u

else return a



Semi-entscheidbar \longrightarrow rekursiv aufzählbar

- f ist total
- f gibt nur Werte aus A aus
- $\forall u \in A \exists k : M$ akzeptiert u in k Schritten
- $f(\text{Kodierung}(k, u)) = u$

■



Äquivalente Aussagen

- A rekursiv aufzählbar
- A semi-entscheidbar
- A ist vom Chomsky-Typ 0
- $A = L(M)$ für TM M
- χ'_A is Turing-, While-, RegM., RAM, ... berechenbar
- A ist Definitionsbereich einer berechenbaren Funktion
- A ist Bereichbereich einer berechenbaren Funktion



2.7 Nicht-entscheidbare Probleme



Normierung von Turing-Maschinen

Betrachte $T = (Q, \Sigma, \Gamma, \delta, s, F)$. OBdA:

- $Q = \{1, \dots, n\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, \sqcup\}, \sqcup = 2$
- $s = 1$
- $F = \{2\}$

für geeignete Konstante n



Gödelnummer $\langle M \rangle$ einer Turingmaschine M

Definiere folgende Zeichenketten über $\{0, 1\}$:

Kodiere $\delta(q, a) = (r, b, d)$ durch $0^q 1 0^{a+1} 1 0^r 1 0^{b+1} 1 0^d$

wobei $N = 1, L = 2, R = 3$ für die Richtungscodierung d gewählt wird.

Die Turing-Maschine wird dann kodiert durch die Binärzahl:

$111\text{code}_1 11\text{code}_2 11 \dots 11\text{code}_z 111,$

wobei code_i für $i = 1, \dots, z$ alle Funktionswerte von δ in beliebiger Reihenfolge beschreibt.

Konvention:

n ist nicht Gödelnummer einer TM,

$\rightarrow n$ beschreibt eine TM, die \emptyset akzeptiert



Beispiel

Sei $M = (\{1, 2, 3\}, \{0, 1\}, \{0, 1, \sqcup\}, \delta, 1, \{2\})$, mit

$\delta(1, 1) = (3, 0, R)$

$\delta(3, 0) = (1, 1, R)$

$\delta(3, 1) = (2, 0, R)$

$\delta(3, \sqcup) = (3, 1, L)$

$\langle M \rangle$ ist dann:

$11101001000101000110001010100100011000100100101000$
 1100010001000100100111



Gödelnummer

Beobachtung

Die Gödelnummerierung beschreibt eine

injektive Abbildung von **normierten** TMs auf natürliche Zahlen



Diagonalsprache L_d

Sei M_i die TM mit $\langle M_i \rangle = i$.

Sei w_i die Binärrepräsentation von i .

$L_d := \{w_i : M_i \text{ akzeptiert } w_i \text{ nicht}\}$

**Satz: L_d ist unentscheidbar****Beweis:**

Annahme:

 $L_d = \{w_i : M_i \text{ akzeptiert } w_i \text{ nicht}\}$ ist entscheidbar.Def. „entscheidbar“
 $\rightarrow \exists M_i : M_i$ akzeptiert L_d und hält stets.Was macht M_i mit w_i ? $w_i \in L_d \xrightarrow{\text{Def. } M_i} w_i$ wird akzeptiert. $\xrightarrow{\text{Def. } L_d} w_i \notin L_d$ $w_i \notin L_d \xrightarrow{\text{Def. } M_i} w_i$ wird nicht akzeptiert. $\xrightarrow{\text{Def. } L_d} w_i \in L_d$ Beides führt zu einem **Widerspruch**.**Korollar:** **$\bar{L}_d = \{w_i : M_i \text{ akzeptiert } w_i\}$ ist unentscheidbar**Annahme: \bar{L}_d ist entscheidbar. $\rightarrow \exists M : M$ akzeptiert \bar{L}_d modifiziere $M \rightsquigarrow M'$ so, dass $M' L_d$ akzeptiert

(Austausch akzeptierende/nichtakzeptierende Haltezustände).

Widerspruch.

**Universelle Turingmaschine** $U = (Q_u, \{0, 1\}, \{0, 1, \sqcup\}, \delta_u, s_u, F_u)$ **Eingabe:** $\langle M \rangle w$ M ist die zu simulierende TM, w ist die **binär codierte** Eingabe. U simuliert M auf w .D.h. U akzeptiert $\langle M \rangle w$ gdw M akzeptiert w **Universelle Turingmaschine**

3 Bänder:

1. $\langle M \rangle$
2. Zustand q_M von M unär codiert
3. Bandinhalt w von M



Universelle Turingmaschine

```

if Präfix  $v$  von  $w$  repräsentiert eine TM then           // 111Tupel111
  verschiebe  $v$  auf Band  $\langle M \rangle$ 
   $q_M := 1$                                                // Startzustand von  $M$ 
  while  $q_M \neq 2$  do                                   // Endzustand von  $M$ 
    laufe zum Anfang von  $\langle M \rangle$ 
    foreach  $(q, a, r, b, d) \in \langle M \rangle$  do           // Feld für Feld
      if  $q = q_M$  then                                   // Vergleich mit Band  $q_M$ 
        if Eingabezeichen von Band 3 =  $a$  then
           $q_M := r$  // auf Zustandsband kopieren
           $b$  auf Band 3 ausgeben
          Bewegung von Band 3 entsprechend  $d$  ausführen

```



Universelle Turingmaschine: 3Band \rightarrow 1Band

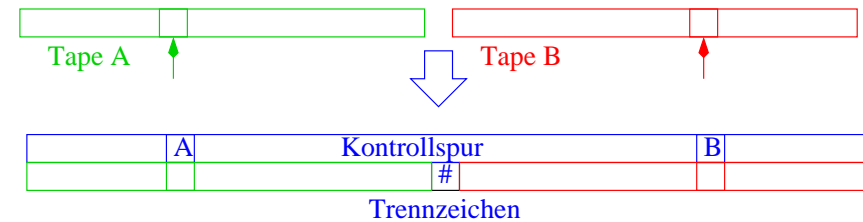
Eigentlich wissen wir wie das geht.

Problem: **Bandalphabet** unabhängig von M aber $> \{0, 1\}$

Codiere Bandalphabet durch konst. viele $\{0, 1\}$.

Problem: **Eingabe** muss auch **codiert** werden.

Das erledigt eine vorgeschaltete **CodierungsTM**.



Halteproblem

$H := \{w_i v : M_i \text{ angesetzt auf } v \text{ hält}\}$

Satz: H ist nicht entscheidbar.

Beweis: angenommen H sei entscheidbar.

Wir konstruieren eine TM, die \bar{L}_d akzeptiert.

$w_i \in \bar{L}_d?$

$\Leftrightarrow M_i$ akzeptiert w_i .

$\Leftrightarrow w_i w_i \in H \wedge M_i$ akzeptiert w_i .

Dies könnten wir mit Hilfe einer **TM für H** und einer **universellen TM** leicht ausrechnen.

Widerspruch.



Das beschränkte Halteproblem

Satz:

$\{w_i v \# w_j : M_i \text{ angesetzt auf } v \text{ hält nach höchstens } j \text{ Schritten}\}$

ist entscheidbar.

Beweisskizze:

Lasse universelle TM U angesetzt auf $w_i v$

für j **simulierte Schritte** laufen.



Mehr unentscheidbare Probleme

Gegeben Turingmaschinen T, T'

- | | |
|--|-----------------|
| <input type="checkbox"/> $L(T) = \emptyset?$ | Leerheit |
| <input type="checkbox"/> $ L(T) = \infty?$ | Unendlichkeit |
| <input type="checkbox"/> $L(T) = \Sigma^*?$ | Vollständigkeit |
| <input type="checkbox"/> $L(T) = L(T')?$ | Äquivalenz |



Unentscheidbarkeit von Leerheit

Angenommen M akzeptiert $\{i : L(M_i) = \emptyset\}$

Wir zeigen, dass \bar{L}_d dann entscheidbar wäre.

$w_i \in \bar{L}_d = \{w_i : M_i \text{ akzeptiert } w_i\}?$

Konstruiere eine Turingmaschine $T(i)$:

erase input

run M_i on w_i

if state(M_i) $\neq 2$ **then** endless loop

Nun ist $L(T(i)) \neq \emptyset$ gdw $w_i \in \bar{L}_d$.

Also wäre \bar{L}_d entscheidbar.

Widerspruch



Unentscheidbarkeit von Vollständigkeit

$L(T) = \Sigma^*?$

Gleicher Beweis wie bei Leerheit! Da $T(i)$ seine Eingabe ignoriert!



Metaprogrammierung

Der Beweis von Leerheit nimmt ein Programm und transformiert es in ein anderes.

Wichtige Programmieretechnik.



Postsches Korrespondenzproblem (PKP)

Geg: endliche Folge von Wortpaaren:

$$K = (x_1, y_1) \cdots (x_n, y_n) \in (\Sigma^+ \times \Sigma^+)^*$$

Frage:

$$\exists i_1, \dots, i_k \in \{1, \dots, n\} : x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k}$$

?



Beispiel

- $K = ((1, 111), (10111, 10), (10, 0))$ hat die Lösung $(2, 1, 1, 3)$,
denn es gilt:

$$x_2 x_1 x_1 x_3 = 101111110 = 101111110 = y_2 y_1 y_1 y_3$$

- $K = ((10, 101), (011, 11), (101, 011))$
hat keine Lösung:
(133...)



Beispiel [Mirko Rahn]

- $K = ((0, 011), (001, 1), (1, 00), (11, 110))$

hat eine kürzeste Lösung der Länge 595:

```

12112121121121212120321211212130321203311213111212031212121121312112
032121121032121303212021111203311213212121213111212112111203203121
2120321211212121213131203213032120320321031213033112131302103201111
1212112111200210121212121203212112121212021203203213032121120321321
3130330321213030312113113032001032121112121131212303212120321210321
0110321303230212123033101120313102121213121020320312021313121321112
10321111212120211112121212032132121211203213031120321121213033121
2131112121203310320312120321213102101032130321020212303302132101100
302121221132031212101032021231321103111212312120303213303003

```



PCP ist semientscheidbar

Algorithmus:

```

Procedure PCP( $(x_1, y_1) \cdots (x_n, y_n)$ )
  for  $k := 1$  to  $\infty$  do
    foreach  $i_1 \cdots i_k \in \{1..n\}^k$  do
      if  $x_{i_1} \cdots x_{i_k} = y_{i_1} \cdots y_{i_k}$  then
        output  $i_1 \cdots i_k$ 
      return

```



PCP ist unentscheidbar

Beweis siehe Schöning.

Idee: angenommen lösbar \rightarrow Halteproblem lösbar

$$x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k} = (s)w\#\dots\#u(f)v$$

beschreibt akzeptierende Folge von TM-Konfigurationen



Hilberts 10. Problem — Diophantische Gleichungen

Gegeben:

multivariates Polynom p

mit ganzzahligen Koeffizienten.

Frage [Hilbert 1900]:



$$\exists x_1, \dots, x_n \in \mathbb{Z} : p(x_1, \dots, x_n) = 0?$$

[Matiyasevich 1970]: Das Problem ist unentscheidbar.



Abgeschlossenheit entscheidbarer Sprachen

abgeschlossen unter

\cup

\cap

\cdot



Abgeschlossenheit **semi**entscheidbarer Sprachen

abgeschlossen unter

\cup

\cap

nicht abgeschlossen unter

\cdot



Abgeschlossenheit **semi**entscheidbarer Sprachen unter Vereinigung

Seien M_1 und M_2 Akzeptoren für L_1 bzw. L_2
Akzeptor für $L_1 \cup L_2$:

for $j := 1$ **to** ∞ **do**

if M_1 accepts w after j steps **then** accept

if M_2 accepts w after j steps **then** accept



Anwendung der nebenläufigen Ausführung

Mehrere Algorithmen A_1, \dots, A_k , die ein schwieriges Problem lösen
(langsam, schnell, nie).

Führe alle Algorithmen (pseudo)gleichzeitig aus.

- Wenn alle gleich schnell haben wir Faktor k Rechenaufwand verschwendet
- + Wenn eins nie fertig wird haben wir unendlich viel gewonnen
- + Bei sehr unterschiedlicher Ausführungszeit können wir im Mittel gewinnen.
- + Wir können parallele Prozessoren verwenden.
- + Oft können wir einen Teil der redundanten Arbeit einsparen



Nichtabgeschlossenheit **semi**entscheidbarer Sprachen unter Komplementbildung

Annahme: Abgeschlossenheit gilt doch.

Sei M Akzeptor für L_d , \bar{M} Akzeptor für \bar{L}_d

Function isInLd(w)

for $j := 1$ **to** ∞ **do**

if M accepts w after j steps **then return** true

if \bar{M} accepts w after j steps **then return** false

Eine von beiden hält.

→ L_d entscheidbar.

Widerspruch



Nebenläufige Ausführung

Anwendungen: Theorembeweiser, Programm/Hardware-Verifizierer,
schwierige Planungs- und Optimierungsprobleme

Beispiel: Erfüllbarkeit aussagenlogischer Formeln in Zeit

$\mathcal{O}((4/3)^{\#\text{Variablen}})$.

[U. Schöningh, A Probabilistic Algorithm for k -SAT and Constraint Satisfaction Problems, FOCS, 1999]